CS 116x Winter 2015
Craig S. Kaplan

Module 10
Data Processing

Topics

- Working with tabular data
- Working with hierarchical data
- Accessing live APIs

Readings

- TBA

Introduction

The modern world is awash in data. We are surrounded by computational devices that generate vast streams of data, more than humanity can possibly consume directly. A simple example is YouTube: last time I checked, YouTube users upload 100 hours of new video to the site every *minute*.

And yet, there's a lot of value to be found in this firehose of data. Games, music, movies, and social media offer endless opportunities for self-edification, if we only knew where to look, how to identify the tiny fraction of content that actually matters to us. Governments and other public institutions offer access to fascinating data streams in which can be found the power for positive societal change.

Because there's too much data to take in directly, we need a way to distill out the interesting bits. A simple human version of this is an infographic, something I

believe GBDA students must learn to create. An infographic is an attempt to reduce a large amount of raw data to an easily digestible nugget that communicates the essence.

Really, though, we need computational tools to process data for us. Terms like Big Data, Data Mining, and Machine Learning are all related to the question of how to extract meaningful trends or unusual occurrences from data. Of course, this science is still young, and in need of better tools. Everyday social media makes this point all too clear: Facebook still shows me a huge pile of junk mixed in with the small set of interesting updates from friends. Dating sites send people out on many, many failed dates before truly compatibles couples find each other (I'm told).

I highly recommend the book *Dataclysm* by Christian Rudder. It's an exploration of interesting trends and statistics that emerge by analyzing the huge volume of data found in OKCupid profiles.

The goal of this module is to introduce the topic of data processing with a few examples that operate on real-world data. Hopefully you will become interested in finding and using data in your own work. We'll talk about two "shapes" of data: tabular data (e.g., spreadsheets) and hierarchical data (through JSON).

Tabular data

For our purposes, a table is a rectangular grid of values, much like a Spreadsheet. The grid has rows and columns. We usually think of each row as defining a single cohesive piece of data, a collection of individual pieces of information that are all related. Each column defines one piece of data associated with each record. In an iTunes music library, the rows are records that describe each song you own; the columns are pieces of information associated with songs (title, artist, genre, etc.). In a marking spreadsheet, the rows correspond to students and the columns to things that were marked (assignments, exams, etc.). Each column may have a *heading* (a name), and a type (which we'll limit to `String`, `float`, and `int`).

A good first place to find interesting tables is in the "open data" websites of public organizations like governments. For example, the Region of Waterloo publishes a catalog of public data in different formats at www.regionofwaterloo.ca/opendata. Let's start there.

Looking through the catalogue of available data, I became curious about "Reserved Street Names", which appears to be available in a simple, easy to read format called CSV ("comma-separated values"). As it turns out, that's a lie. The CSV link opens up a text file with cute boxes drawn using ASCII Art:

```
-----------------------------------------
|   FullStreetName   |    Municipality    |
-----------------------------------------
| Abbey Glen         | Kitchener          |
-----------------------------------------
| Aberle             | Woolwich           |
-----------------------------------------
| Abeth              | Kitchener          |
-----------------------------------------
| Abitibi            | Cambridge          |
-----------------------------------------
| Able               | Cambridge          |
-----------------------------------------
| Abram Clemens St   | Kitchener          |
```

OK, yes, they also make the same data available in an Excel spreadsheet, and I could easily export a real CSV file from Excel. But this turns out to be a nice lesson in handling messy real-world data. We should be able to write some Processing code to read this file despite the extraneous ASCII Art. We can use the Text Processing tools we learned previously to extract the useful bits from this file:

```
void setup()
{
  String[] lines = loadStrings(
    "ReservedStreetnames.txt" );
  for( int idx = 0;
       idx < lines.length; ++idx ) {
    if( lines[idx].startsWith( "-" ) ) {
      // Line starts with -, just skip it.
      continue;
    } else {
      // Extract the text between the vertical pipe
      // characters
      String[] boxes = splitTokens(
        lines[idx], "|" );
      // Use the trim() function to remove extra
      // whitespace around every street name and
      // municipality
      boxes = trim( boxes );
      println( "{" + boxes[0] + "} "
        + "{" + boxes[1] + "}" );
    }
  }
}
```

This example doesn't really do anything. It just prints out the street names and municipalities, surrounding them with curly braces to prove that they've been extracted correctly. A better demonstration of capturing this data would be to store it in some kind of table data structure. Sure enough, Processing gives us a `Table` class for exactly this purpose. We can create `Table` instances, add and remove rows and columns, and populate the table with data. The online documentation for `Table` gives a good example of doing this, and you can play with a sample sketch that constructs a `Table` from this dataset.

Example sketch: ReservedStreetsTable

Honestly, there isn't all that much that can be done with this data, but it's a good starting point. Let's move on to something more interesting: money. The province of Ontario requires that the salaries of all public sector employees who make over $100,000 a year be made available to the public. The data is all online. Again, though, it's not in a particularly

convenient format: you can look at it as HTML on a web page, or you can download a PDF. That's a good starting point, but what if I want to perform real calculations on the data? For example:

- Who is the most highly paid person in a given sector in the province? The most highly paid at a given institution?
- What is the average salary for all the people with a given job title?
- What's the most extreme salary inversion, e.g., the highest paid junior professor?

It would be useful to grab this data and get into a format that permits computation. Computer scientists sometimes talk of "scraping", "snarfing", or "munging" data. It's not obvious, but there's an easy way to do that—simply highlight the entire table on one of these HTML pages and copy it into a text file. My Chrome browser inserts tabs at all column breaks in the table. We can use the tab character (written '\t' in Processing) as a delimiter to break lines of the table into fields:

```
void setup()
{
  Table table = new Table();
  table.addColumn( "Institution" );
  table.addColumn( "Last name" );
  table.addColumn( "First name" );
  table.addColumn( "Position" );
  table.addColumn( "Salary", Table.FLOAT );

  String[] lines = loadStrings( "salaries.txt" );
  for( int idx = 0; idx < lines.length; ++idx ) {
    String[] fields = splitTokens(
      lines[idx], "\t" );
    TableRow row = table.addRow();
    row.setString( "Institution", fields[0] );
    row.setString( "Last name", fields[1] );
    row.setString( "First name", fields[2] );
    row.setString( "Position", fields[3] );
    row.setFloat( "Salary",
      processDollars( fields[4] ) +
      processDollars( fields[5] ) );
  }
}
```

The function `processDollars()` is a helper function that converts a `String` into a floating-point dollar amount. See the sample sketch for the implementation.

Example sketch: Salaries

That's still a hefty amount of work to read in a file. We can do something very slick with this file, though. Recall that the fields are delimited by tab characters. It turns out that "tab-separated values", or TSV, is a standard file format for tables, one that Processing understands natively. So we can replace almost all of the above with a single line:

```
Table table = loadTable( "salaries.txt", "tsv" );
```

The `loadTable()` function behaves a lot like `loadStrings()` or `loadImage()`, as you might expect. The optional second argument gives hints to Processing for how to interpret the tabular data. In this case we inform Processing that the file is in TSV form instead of the usual CSV (comma-separated values). There's one downside here, namely that the columns don't get names when we do this. That means we have to refer to them by position instead:

```
// You can't do this anymore, because Processing
// doesn't know about the names of the columns.
// String str1 = table.getString( 5, "Position" );

// This works.
String str2 = table.getString( 5, 3 );
```

But we can fix this with a small tweak. We can manually add a new first row to the input file containing the names of the columns. Processing supports another option to treat the contents of the first row as special:

```
Table table = loadTable(
  "salaries.txt", "header, tsv" );
```

One final problem is that we want to collapse the two final columns of the table in the source file into a single column containing the sum of salaries and benefits. The sample sketch shows how to do this.

Be sure to look at the beginning of the `salaries.txt` file in the **SalariesTable** sketch, to see the addition of the column headers.

Example sketch: SalariesTable

As a final example of reading tabular data, consider the Region of Waterloo's food inspection reports. This dataset comes in three separate tables:

- A *Facilities* file, in which each row gives complete information about a single food-serving facility in the region. The first column is a unique ID code associated with each facility, which will be used in the other tables to refer to it.
- An *Inspections* file, a table that lists individual inspections in which someone visited a facility. Each inspection has its own unique ID, and mentions the ID of the facility to record where the inspection took place.
- An *Infractions* file. Each inspection may result in zero or more infractions against the food safety code. These infractions are recorded one per row in this file. The infraction record refers back to the inspection ID.

The good news is that each of these files is in a proper CSV format, and can be read into a sketch in a single line of code. The difficulty is that in order to do interesting things, you have to gather information from across multiple tables. For example, here's how to list all the infractions associated with a given restaurant, given the restaurant's name:

- Iterate over the rows of the Facilities table. For each row, check if the "BUSINESS_NAME" column matches the name you're looking for. If it does, save the ID associated with that name.
- Iterate over the rows of the Inspections table. For each row, if the "FACILITYID" column matches the

facility you're looking for, append the associated inspection ID to an array of strings.
- Finally, gather together an array of all the infractions in the Infractions table whose "INSPECTION_ID" is in the list of IDs set aside in the previous step.

Wow, that's a lot of work. It's more than I expected would be necessary when I started playing with the dataset, and more than I would ever ask for in this course. But the result is fairly cool—a sketch in which you can type in the name of a restaurant and see a complete list of its infractions. That was sufficiently worthwhile that I decided to include it here. (It's still only a bit over 100 lines of code.)

Example sketch: FoodInspections

Before I leave this section of the notes, let me offer a summary list of useful table-related functions in Processing. See the `Table` documentation for complete information.

Creating a table

- `new Table()`

Filling a table

- `addColumn(), addRow()`
- `setInt(), setFloat(), setString()` (these exist as three-argument methods of `Table`, or two-argument methods of `TableRow`)

Reading a table

- `getRowCount(), getColumnCount()`
- `getRow()`
- `getInt(), getFloat(), getString()` (these exist as two-argument methods of `Table`, or one-argument methods of `TableRow`)

Hierarchical data

Sometimes we want to obtain data from the outside world that isn't as cleanly structured as an array of objects or a table. For example, we might want to load in all of the information about a restaurant. That data might include a wide array of heterogeneous data:

- The name, address and phone number of the restaurant (as strings)
- The opening hours, which could be an array of seven objects, each of which is made up of two strings (the opening time and closing time each day of the week). Or maybe each time is given as two integers, and hour and a minute
- A list of strings giving links to review sites
- A set of menus (breakfast, lunch, dinner), each of which contains a heading describing the menu together with an array of records giving names, descriptions and prices of dishes.
- etc.

Data shapes like arrays and tables are good for lots of applications (including sub-parts of our hypothetical restaurant information), but they just can't handle this kind of freeform structured data in full. There are numerous ways that this kind of data does get represented in practice. Two very popular forms are XML (eXtended Markup Language, which we won't talk about in this course) and JSON (JavaScript Object Notation, which will form the rest of the module).

JSON is a very small subset of the Javascript language, which can be used to describe collections of data. It started out as a means for a script running on a web page to exchange data with a web server. But it was so simple and useful that it became a bit of a standard way for programs to send structured information back and forth. In particular, it's built in to Processing.

Loading JSON objects

The simplest way to get a JSON object into a sketch is to load it from a file. The format should be familiar by now, as it's analogous to reading images, vector illustrations and tables.

```
JSONObject my_obj =
  loadJSONObject( "data.json" );
```

Reading data from JSON objects

A `JSONObject` behaves a lot like an instance of a class. It has a number of named *fields*, and each field has an associated type, and stores a value. There are six types that we'll need to think about for fields. A field can have one of the familiar types `int`, `float`, `boolean` or `String`. A field can also contain an array, which is stored using the class `JSONArray`, or it can even contain a nested (smaller) `JSONObject`.

However, a `JSONObject`'s fields aren't treated like class fields by Processing. So after loading in `my_obj` above, you can't say something simple like `my_obj.fieldname` as you might with a regular class instance. Instead, you need to call a method of `my_obj` that reads the contents of the field for you. If you knew that your JSONObject had a field called "name" of type `String` and a field called "weight" of type `int`, then you could write code like this:

```
String obj_name =
  my_obj.getString( "name" );
int obj_weight =
  my_obj.getInt( "weight" );
```

A `JSONObject` can contain another `JSONObject` in one of its fields, and it can contain a `JSONArray` in one of its fields. Similarly, a `JSONArray` can contain a `JSONObject` or another `JSONArray` in any of its numbered entries. There's nothing magical about this, but it just means that there may be cases where you need to attach a few calls to, e.g., `getJSONObject()`

together to “drill down” to the lowest level of a chunk of hierarchical data:

```
JSONArray my_arr =
  my_obj.getJSONArray( "data" );
JSONObject my_event =
  my_arr.getJSONObject( 0 );
String name =
  my_event.getString( "name" );
```

...or, if you’re feeling a bit more intense, you can do all of this in one statement, borrowing a bit of unusual syntax from ControlP5:

```
String name = my_obj
  .getJSONArray( "data" )
  .getJSONObject( 0 )
  .getString( "name" );
```

You almost certainly won’t be constructing `JSONObject` instances from scratch, so once you’ve acquired an instance by loading it in, you basically just need to use the methods `getString()`, `getInt()`, `getFloat()`, `getBoolean()`, `getJSONArray()`, and `getJSONObject()`. Each of these methods takes a single `String` as its argument, corresponding to the name of the field you want to retrieve. The `JSONArray` class supports exactly the same methods, except in that case they take an `int` as an argument, corresponding to the location you want to read from the array.

The easiest way to tell what fields a JSON object supports is to read the documentation provided by whoever gave you the object. If that doesn’t work, it’s helpful to look at the object itself (i.e., the raw input file)—they’re pretty easy to read.

Example sketch: SimplestJSON

Web APIs

An API (Application Programming Interface) is a fancy software engineering name for something we've dealt with all term. It's the set of functions that a given library understands, through which you access its features. So far this term, all the APIs we've used have either been built in to Processing, or accessible through an `import` statement.

But here's the exciting bit—many online services offer APIs as well! We use many online services these days that organize vast amounts of data on our behalf. Social media certainly works this way (Facebook stores piles of status updates, photos, notes, lists of friends etc.), as do photo-sharing sites, cloud-based storage, and other information resources. Some of these services publish public APIs through which you can access the underlying data without having to knock on the front door by visiting the service with a web browser. This is a very powerful idea: it lets you write you own custom applications that use a pile of data without getting stuck with the service's view of how that data should be viewed.

You can think of accessing a Web API as "calling a function over the internet". You're calling a function in order to obtain some piece of information as a result, but instead of your computer working out the answer to the function on its own, it sends the request off to a second computer. That computer figures out the answer and ships it back to you in the form of a `JSONObject` (or some other structured data value).

In order to call the function you want, you need to package up your request into a form that can be sent off to the other computer. The nice bit is that this doesn't require any new ideas or code. Web API calls look just like URLs, and you can "call the function" by giving the URL you want to `loadJSONObject()`.

A good source of examples is api.uwaterloo.ca, the University of Waterloo's own internal open data API. It supports a number of different function calls that return information about the campus and its environs. Visit the API's documentation page to see the list of queries you can make. If you want to find out the current weather, for example, you can access the following URL: https://api.uwaterloo.ca/v2/weather/current.json. Try it now in your browser! Hopefully it will show you the text of the JSON as output (it works in Chrome, at least). So, if you store that JSON object in a variable:

```
JSONObject weather =
  loadJSONObject( "https://api.uwaterloo.ca/v2/weather/current.json" );
```

Then you can start querying the variable `weather` to find out things like the current temperature and precipitation. (The final assignment of the course is very much like this, but I give you a class that handles the querying for you. You ask that class to give you back the `JSONObject`.)

If you try other UW API calls, you'll find that they return an error object saying that you need an API key. Most Web APIs require you to pass in a key along with your request. The key identifies you, and allows the web service to track how much you're using their data (which is useful, for example, if they want to bill you for your use of their service). In the assignment, you'll see that I've provided a single key for the UW API that you can use for the duration of the term. Normally you'd register for your own key as a developer.

If you try searching the internet for the name of your favourite online service together with "API", you'll see the range of tools available to programmers. Twitter's API is a particularly well known one. Facebook has one for the graph of your connections to your friends, but not for status updates. The Google Maps API powers a large number of online tools by third parties. Even IMDB and Rotten Tomatoes publish APIs for getting at live movie data. Many mobile apps are really just user interfaces wrapped around accesses to

Web APIs. Waterloo's getting in on the game too. The current API will soon be enhanced with a lot more functionality and pretty soon you'll be invited to use the new Waterloo Student Portal, which runs on top of that API layer.